

OCL and Graph-Transformations – A Symbiotic Alliance to Alleviate the Frame Problem*

Thomas Baar

École Polytechnique Fédérale de Lausanne (EPFL),
School of Computer and Communication Sciences,
CH-1015 Lausanne, Switzerland
`thomas.baar@epfl.ch`

Abstract. Many popular methodologies are influenced by Design by Contract. They recommend to specify the intended behavior of operations in an early phase of the software development life cycle. Formal contract specification languages, however, are still rarely used because their semantics often mismatch the needs of software developers. Restrictive specification languages usually suffer from the "frame problem": It is hard to express which parts of the system state should remain unaffected when the specified operation is executed. Constructive specification languages, instead, suffer from the tendency to make specifications deterministic. This paper investigates how a combination of OCL and graph transformations can overcome the frame problem and can make constructive specifications less deterministic. Our new contract specification language is considerably more expressive than both pure OCL and pure graph transformations.

Keywords: Design by Contract, Behavior Specification, Graph Grammars, OCL, QVT.

1 Motivation

Design by Contract (DbC) [1, 2] encourages software developers to specify the behavior of class operations in an early phase of the software development life cycle. Precise descriptions of the intended behavior of operations can be of great help to grasp design decisions and to understand the responsibilities of classes identified in the design. The specification of behavior is given in form of a *contract* consisting of a pre- and a postcondition, which clarify two things: The pre-condition explicates all conditions that are expected to hold whenever the operation is invoked. The post-condition describes how the system state looks like upon termination of the operation's execution.

There are many specification languages available to define contracts formally. Despite their differences at the surface level, all languages can be divided into only two classes. The classification is based on the technique to specify the

* This work was supported by HASLER-Foundation, project DICS-1850.

post-condition of a contract. *Restrictive specification languages* formulate the post-condition in form of a predicate, i.e. a Boolean expression, which restricts the allowed values for properties in the post-state. Well-known examples for restrictive languages are OCL, JML, Z, and Eiffel. *Constructive specification languages* interpret post-conditions not as restrictions on the post-state but – conceptually completely different – as updates, which transform the pre-state into the post-state. In many cases, an update denotes a deterministic contract: for a pre-state that satisfies the pre-condition the post-state can be computed deterministically. Well-known examples for constructive languages are B, ASM, graph transformations, and UML’s Action Language.

This paper investigates how the expressive power of constructive languages – as an example we consider one form of graph transformations – can be improved to master non-deterministic contracts. In Sect. 3, the basic elements of graph transformations are extended with restrictive specification elements (OCL clauses). In its extended version, graph transformations are more powerful but still not powerful enough to formalize all contracts that are relevant in practice. Thus, a second extension is discussed in Sect. 4, which allows to simulate the loose semantics of restrictive languages. To summarize, the proposed extensions of graph transformations enable software developers to write formal contracts that (1) do not suffer from the frame problem, (2) are non-deterministic, and (3) allow to change a state freely.

Related Work. The idea to use graph transformations to formalize contracts is not novel. There are even already tools for this purpose available [3, 4]. The examples we found in the literature, however, are always deterministic contracts, which do not require to extend graph transformations with restrictive specification elements.

The idea to extend graph transformations with OCL clauses has been adopted from the Query/Views/Transformations proposal (QVT) [5], which is a response on a corresponding request for proposals by the OMG. In Sect. 3, the QVT approach is put into a broader context by providing the link from model transformation (the original application domain of QVT) to formal contract specification.

Extending graph transformation rules with OCL also means to combine OCL with object diagrams, what has been explored in the literature also for a different target than contract formalization. The language VOCL (Visual OCL) uses collaborations to represent OCL constraints in a visual format for better readability [6]. Similarly, the proposal made by Schürr in [7] is inspired by Spider diagrams and aims at a more readable, graphical depiction of OCL constraints. The approaches described in [6, 7] cannot be compared with the approach presented in this paper because they have a fundamentally different goal. Firstly, [6, 7] do not use OCL in order to improve the expressive power of a graphical formalism. Instead, the graphical formalism is merely used as an alternative to OCL’s textual standard syntax. Secondly, our approach targets only operation contracts whereas [6, 7] aim at a visualization of any kind of OCL constraints including invariants.

2 Restrictive Languages and the Frame Problem

2.1 Example: CD Player

The main purpose of CD players is to entertain people and to play the content of compact discs (CDs). The content of a CD is organized by tracks that are burned in a certain order on the CD. We want to assume that a CD can be played in two modes. In the normal mode, all tracks on the CD are played in the same order as they appear on the CD. In addition, the CD player can work in a shuffle mode in which the tracks are played in a randomized order. Finally, we want to assume that a CD player has a display on which, depending on the chosen display mode, the elapsed or remaining time for the current track is shown. This CD player scenario is modeled straightforwardly by the class diagram shown in Fig. 1.

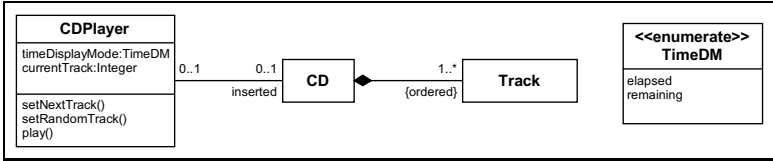


Fig. 1. Static model of CD player scenario

In the next subsection, we will focus on the formal behavior specification for the operations `setNextTrack()` whose intended semantics is to determine the next track to be played if the CD player is working in the normal mode. The operation `setRandomTrack()` will be specified in Sect. 3 and determines the next track if the CD player works in the shuffle mode.

2.2 Complexity of the Frame Problem

The intended semantics of operation `setNextTrack()` is to move one track forward on the CD and to increase the value of attribute `currentTrack` by one. The formalization of this behavior in a restrictive language such as OCL seems to be straightforward but there are some traps one can fall into.

```

context CDPlayer::setNextTrack()
  pre: self.inserted ->notEmpty()
  post: self.currentTrack = (self.currentTrack@pre mod
    self.inserted.track->size()) + 1
  
```

This contract has some merits since it resolves ambiguities that were hidden in the informal description of the behavior. The first important information is expressed by the pre-condition saying that the CD player assumes to have a CD inserted whenever the operation `setNextTrack()` is invoked. Note that this assumption is indeed necessary because the post-condition navigates over the

currently inserted CD. The second merit of the contract is to make explicit the behavior of `setNextTrack()` when the current track is the last one on the CD. Reasonable variants might be to set `currentTrack` to zero (and thus to stop playing) or to continue with the first track on the CD as it is stipulated by our OCL constraint.

Although the OCL contract clarifies the informally given specification in some respects, it does not capture completely the intended behavior. According to the formal semantics of OCL in [8], an implementation still fulfills the contract even if it would not only change the value of `currentTrack` but also the display mode (attribute `timeDisplayMode`). Or the implementation could create/delete other objects, or could change the state of other objects, or could change the connections (links) between objects.

3 Constructive Languages and Non-deterministic Contracts

Graph transformations are introduced as a constructive specification language. It is discussed, why pure graph transformations are able to specify the operation `setNextTrack()` but fail to specify `setRandomTrack()` correctly. To overcome this problem, we finally discuss a combination of constructive and restrictive specification style.

3.1 Non-deterministic Contracts

Non-deterministic contracts are necessary when not all details of the operation behavior should be fixed in time of writing the contract.

The intended behavior of `setRandomTrack()` is a typical example for a non-deterministic contract. The operation name `setRandomTrack` might be misleading as it might set up the expectation that our contract will enforce a true randomized behavior of the implementation in the sense that invoking the operation twice in the same state will most likely result in different post-states. Note that this kind of randomness cannot be expressed by a contract (neither in OCL nor in any other contract language) because it would require to describe formally the behavior of multiple invocations whereas a contract can specify only the behavior of a single invocation.

The specification of `setRandomTrack()` in OCL looks as follows:

```
context CDPlayer :: setRandomTrack()
pre: self.inserted ->notEmpty()
post: Set{1..self.inserted.track->size()}
      ->includes(self.currentTrack)
```

This contract suffers again from the frame problem but, if this is ignored for a while, the post-condition keeps intentionally the exact post-state open and thus allows many different implementations. Even, an implementation that constantly sets attribute `currentTrack` to 1 was possible and would conform to this contract.

3.2 Graph Transformations as a Constructive Language

A graph transformation rule consists of two graph patterns called left-hand side (LHS) and right-hand side (RHS). Graph patterns are normal graphs whose elements, i.e. nodes and links connecting some nodes, are identified by labels.

Besides this basic version of graph transformation rules, modern graph transformation systems offer much more sophisticated elements to describe patterns such as typed nodes, multiobjects, negative application conditions (NACs), parameters, etc. (see [9]). In the rest of the paper, we will use the graph transformation system QVT submitted as a proposal to the OMG for the standardization of model transformations. For details on the syntax/semantics of this formalism, the interested reader is referred to [5]. A bigger example on how QVT can be used as a contract specification language is given in [10].

As a simple example for a behavioral specification using graph transformations, Fig. 2 shows a rule specifying the intended behavior of `setNextTrack()`.

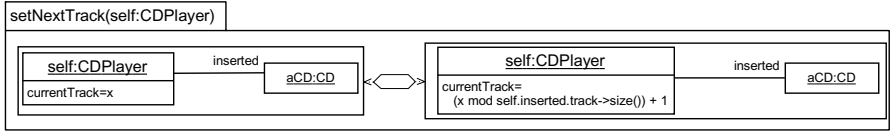


Fig. 2. Specification of `setNextTrack` with QVT

The graph patterns LHS, RHS use typed nodes (e.g. `self:CDPlayer`) that must conform to the system description given in Fig. 1. The LHS of the rule serves two things. First, it imposes restrictions that must hold in order to make the rule applicable for the given state. For `setNextTrack()`, the effective restriction is that the CD player `self` has a CD inserted (expressed by the link between `self` and `aCD`). The second purpose of LHS is to query the pre-state and to extract information that is important for the post-condition encoded by RHS. In our example, the variable `x` extracts the current value of attribute `currentTrack`. Note that the attribute `currentTrack` could have been omitted in LHS and the rule would still be applicable on exactly the same set of graphs as before (but, in this case, RHS had to be reformulated).

The RHS of `setNextTrack()` is almost identical to LHS except for the value of attribute `currentTrack`. Consequently, applying the rule on a state will change only the value of `currentTrack` on the object `self` and nothing else. The new value of this attribute is computed based on the information queried during the first step of the rule application.

3.3 Mixing Constructive and Restrictive Languages

Graph transformation rules, as they were explained so far, can capture deterministic contracts in an elegant way whereas it seems hopeless to use them for non-deterministic contracts.

Fortunately, there is a solution and the same problem has been already tackled by other constructive languages. The language B, for example, offers, besides a pseudo-programming language for computing the post-state, the construct ANY-WHERE. This construct causes a non-deterministic split in the control flow and connects the same pre-state with possibly many post-states. The non-deterministic choices are, however, restricted by a predicate, which has to be evaluated in all control flows to true. In other words, constructive and restrictive specification style is mixed. The formal semantics of ANY-WHERE is defined in [11]. For an example-driven explanation of ANY-WHERE, the reader is referred to [12].

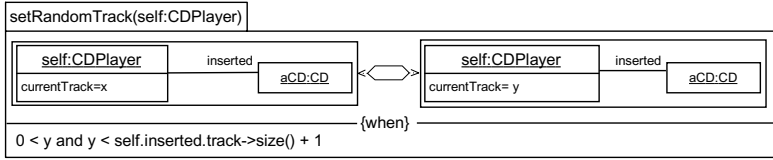


Fig. 3. Specification of `setRandomTrack` with QVT

Basically, for increasing the expressive power of graph transformations the same idea as in B can be applied. In QVT, variables can occur in RHS even if they do not occur in LHS. Consequently, the value of these fresh variables is not fixed anymore by the first step of the rule application and can be chosen non-deterministically. In order to get at least partial control over the values of these variables, QVT has added when-clauses to transformation rules. A when-clause contains constraints written in OCL. The constraint restricts the possible values not only for fresh variables used in RHS but for all elements in LHS and RHS.

The specification of `setRandomTrack()` shown in Fig. 3 takes advantage of the fresh variable `y` in RHS. The value of `y` is restricted in the when-clause what exactly captures the intended semantics.

4 Giving Graph Transformations a Loose Semantics

Although the integration of the when-clause is a necessary step to make graph transformations widely applicable and to overcome the determinism problem, this step is not sufficient. Another immanent problem of constructive languages remained unsolved. It is sometimes necessary to express in the contract that the implementations of the operation are allowed to change parts of the system state in an arbitrary way. If one puts this request to its very end, it means that in some cases the loose semantics of restrictive languages is needed.

In this section, we propose an extension of QVT that makes it possible to simulate the loose semantics of purely restrictive contracts written in OCL. These enrichments require a slight extension of QVT's notation to describe LHS and RHS.

4.1 Possible Side Effects of Restrictive Specifications

As argued in Sect. 2, the contract for `setNextTrack()` written in OCL does not exclude unintended side effects. These side effects can be classified as follows:

1. On object `self`, the values of the attributes not mentioned in the post-condition might have been changed.
2. The values of attributes of `CDPlayer`-objects different from `self` might have been changed.
3. The values of attributes of objects of other classes might have been changed.
4. An unrestricted number of objects of some classes might have been newly created.
5. An arbitrary number of existing objects except `self` might have been deleted.
6. An arbitrary number of links might have been created/deleted.

We will demonstrate in Sect. 4.3 how the contract for `setNextTrack()` shown in Fig. 2 had to be changed in order to capture each of these possible side effects. Beforehand, in the next subsection, the new constructs proposed for QVT, which are needed to simulate loose semantics, are summarized.

4.2 A Proposal for Extending QVT

Optional Creation/Deletion of Objects and Links. Graph transformation rules must be able to express that an object is optionally created or deleted. The same holds for links. So far, one can only specify that an object/link must have been created (deleted) by displaying the object/link in RHS but not LHS (in LHS but not in RHS). We propose to adorn an object/link in RHS with a question mark ('?') to mark its optional creation/deletion.

Note that it is a proven technique to adorn elements in LHS and RHS in order to modify the standard semantics of the rule. QVT and other graph transformation formalisms allow already to adorn elements with 'X' in order to express a negative application condition (NAC).

Placeholders to Denote Arbitrary Attributes/Classes. A more significant extension of graph transformations is the introduction of placeholders. Currently, QVT allows to describe the change of an attribute value only if the name of the attribute is known. One can, for example, not specify the reset of all attributes of type Integer to 0 unless all these attributes explicitly occur in the graph transformation rule.

We propose to use placeholders for attributes as a representation of arbitrary attributes. These placeholders appear in the same compartment of the object as normal attributes. In order to distinguish between normal attributes and placeholders, we start the name of the latter always with a backslash (\). This convention relies on the assumption that the name of normal attributes never starts with backslash. For example, if `\att` appears in the attribute compartment of an object, then it represents all attributes of this object (including attributes inherited from super-classes).

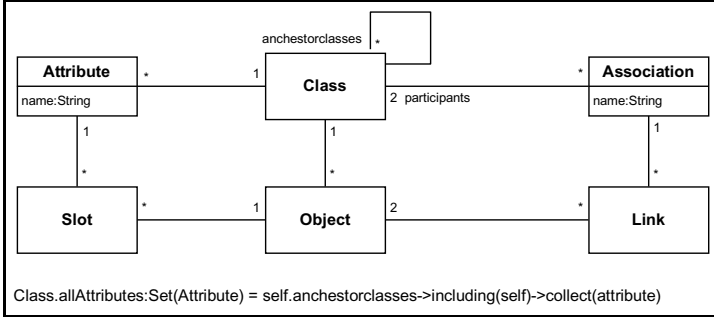


Fig. 4. Simplified metamodel for states

Sometimes, a placeholder should not represent all possible attributes but only some of them. To achieve this, we propose to use QVT’s when-clause to define using OCL constraints which attributes are represented by which placeholders. Such OCL constraints, however, refer to the metamodel of UML object diagrams. To ease the understanding, we rely here on a simplified version of the official metamodel as shown in Fig. 4.

Furthermore, in order to distinguish easily OCL constraints referring to the metamodel from ordinary ones, we decided – slightly abusing OCL’s official concrete syntax – to precede within OCL expressions each navigation on the metalevel with a backslash.

Besides placeholders for attributes there are also analogously defined placeholders for classes.

4.3 Realization of Possible Side Effects

We give examples on how the side effects of OCL constraints presented in Sect. 4.1 can be simulated using our extension of QVT. In all cases, we start from the constructive specification of `setNextTrack()` shown in Fig. 2.

Other Attributes for Self Can Change. A naive solution could be to explicitly list all attributes of object `self` in both LHS and RHS and to assign in RHS a fresh variable to the attribute.

This solution is first of all tedious to write down and in addition has the limits that were already discussed: In time of writing the contract, not all subclasses of `CDPlayer` might be known. Be aware that the QVT rule formulated in Fig. 2 is applicable even when `self` matches with an object whose actual type is not `CDPlayer` but a subclass of it. The core of the problem is, that, when writing the contract, we cannot predict which attributes the object `self` actually has.

The rule shown in Fig. 5 overcomes this principal problem. Each attribute of `self` is represented by placeholder `\attDiffCurrentTrack` as long as its name is different from ‘currentTrack’. This is precisely described in the when-clause by an OCL constraint: For the actual class of `self` (which might be a subclass

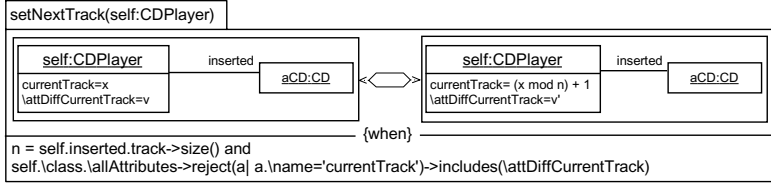


Fig. 5. Different attribute values for **self**

of **CDPlayer**) all valid declarations of attributes (including declarations from super-classes) are collected. The OCL constraint in the when-clause stipulates that the placeholder `\attDiffCurrentTrack` stands for any attribute as long as it is not named 'currentTrack' since attribute `currentTrack` cannot be changed in an arbitrary way. The value of `\attDiffCurrentTrack` in LHS is represented by variable `v`, which does not occur in the RHS. The new value `v'` in RHS shows that the value of the attribute matching with `\attDiffCurrentTrack` might have been changed during the execution of the operation.

State of Other CDPlayer-Objects Might Change. This side effect is similar to the effect of changing the state of **self** and can be captured by applying the same technique to enrich the QVT transformation. A new object **other** is added to both LHS and RHS. In RHS, the value of the placeholder `\att` is changed to a possibly new value `v'`.

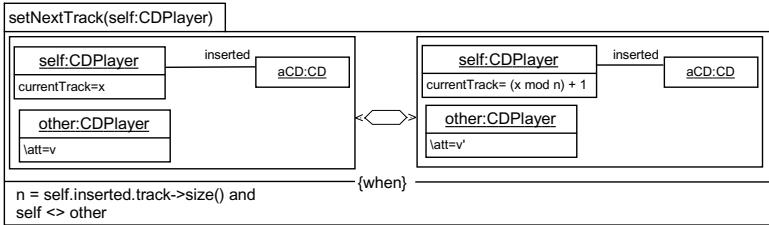


Fig. 6. Different attribute values for other objects of class **CDPlayer**

State of Objects of Other Classes Might Change. In order to simulate state changes on objects of arbitrary classes different from **CDPlayer** (and its subclasses) placeholders for classes are needed. We have introduced the placeholder `\OtherClass` whose value is restricted by an appropriate constraint in the when-clause. The technique to change the state of objects of class `\OtherClass` is the same as the one exploited above to simulate the state change of **CDPlayer**-objects.

Objects Different from Self Might Have Been Deleted. It is not enough to add the question mark to the new object **other** (that represents an arbitrary

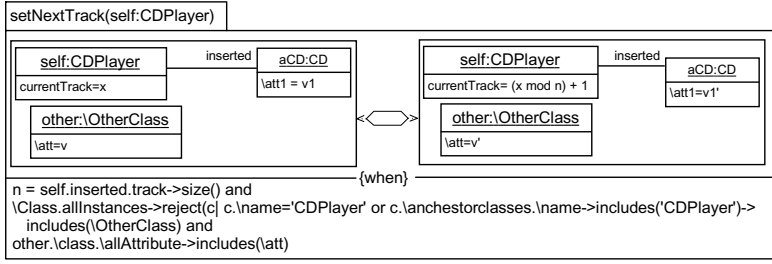


Fig. 7. Different attributes for object of other classes

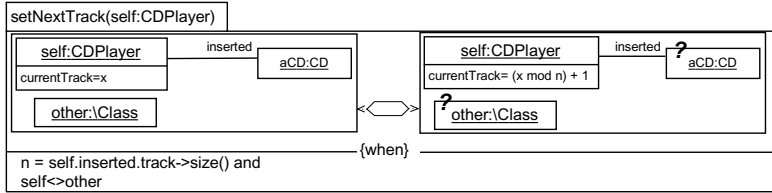


Fig. 8. Deletion of objects

object different from **self**). Unfortunately, the question mark must also be attached on all objects different from **self** that are explicitly mentioned in RHS (without such a question mark, the QVT semantics stipulates that all objects occurring in RHS are not deleted).

Objects Might Have Been Created. Optional creation of arbitrarily many objects is expressed by adding a multiobject **other** to RHS. For each class, **other** represents the set of newly created objects.

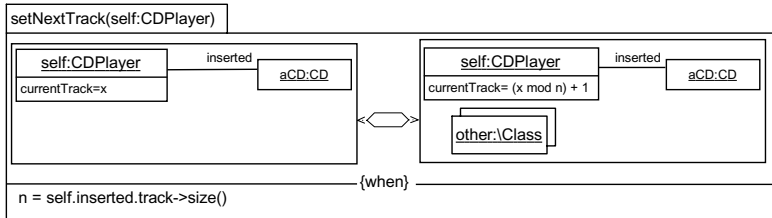


Fig. 9. Creation of objects

Links Might Have Been Created. For the optional creation of links, two arbitrary objects **o1**, **o2** are searched in LHS. The classes of **o1**, **o2** must be connected by an association with name **assoname**. RHS stipulates the optional creation of a corresponding link between both objects.

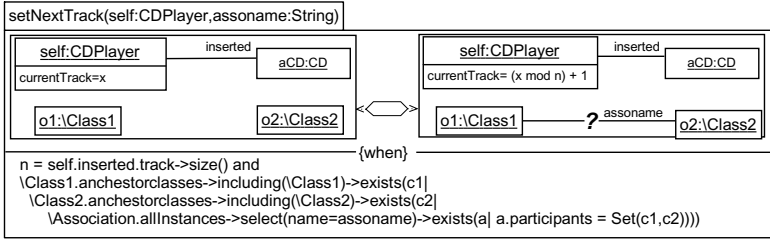


Fig. 10. Creation of links

Links Might Have Been Deleted. Analogously to the optional deletion of objects we mark also links that are deleted optionally with a question mark.

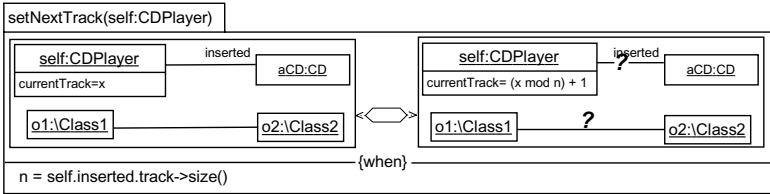


Fig. 11. Deletion of links

5 Conclusion and Future Work

In this paper, pros and cons of the two main behavior specification paradigms – constructive and restrictive style – are discussed. If restrictive languages do not provide provision for tackling the frame problem (such as OCL), then the specified contracts are comparably weak and do most often not capture the behavior intended by the user. Constructive languages suffer from the opposite problem as they sometimes prescribe too detailed the behavior and do not allow the freedom for variations among possible implementations. These two fundamental problems make it also very difficult to define a semantically preserving transformation from specifications of restrictive specification languages into specifications written in a constructive language, or vice versa.

Graph transformations can be used as a basically constructive specification language but it is sometimes also possible to pursue a restrictive specification style. Contracts given in form of a graph transformation rule have the advantage of being easily accessible by humans due to the visual format. In many cases, constructive contracts are intended and constructive contracts work well. For the case that a purely constructive semantics is not appropriate, we have given in Sect. 4 a catalog of proposals to enrich a graph transition rule so that the intended behavior is met. This approach to adapt the semantics of the rule more to the loose semantics of restrictive languages is very flexible since the user has the possibility to traverse the metamodel with OCL constraints.

A lot of work remains to be done. First of all, the proposed formalism of extended graph transformations should be implemented by a tool to resolve all the small problems that can only be recognized if a tool has to be built. In order to become confident in the formal semantics of the formalism, an evaluator needs to be implemented that can decide for any contract and any given state transition whether or not the transition conforms to the contract.

Once such a tool is available, it should be applied on bigger case studies showing or disproving the appropriateness of the proposed formalism for practical software development.

References

1. Bertrand Meyer. Applying “design by contract”. *IEEE Computer*, 25(10):40–51, October 1992.
2. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, second edition, 1997.
3. Claudia Ermel and Roswitha Bardohl. Scenario animation for visual behavior models: A generic approach. *Software and Systems Modeling (SoSym)*, 3(2):164–177, 2004.
4. Lars Grunske, Leif Geiger, Albert Zündorf, Niels van Eetvelde, Pieter van Gorp, and Dániel Varró. *Model-driven Software Development - Volume II of Research and Practice in Software Engineering*, chapter Using Graph Transformation for Practical Model Driven Software Engineering. Springer, 2005.
5. OMG. Revised submission for MOF 2.0, Query/Views/Transformations, version 1.8. OMG Document ad/04-10-11, Dec 2004.
6. Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. Consistency checking and visualization of OCL constraints. In *UML 2000 - The Unified Modeling Language*, volume 1939 of *LNCS*, pages 294–308. Springer, 2000.
7. Andy Schürr. Adding graph transformation concepts to UML’s constraint language OCL. *Electronic Notes in Theoretical Computer Science, Proc. of UNIGRA 2001: Uniform Approaches to Graphical Process Specification Techniques*, 44(4), 2001.
8. OMG. UML 2.0 OCL Specification – OMG Final Adopted Specification. OMG Document ptc/03-10-14, Oct 2003.
9. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
10. Slaviša Marković and Thomas Baar. Refactoring OCL annotated UML class diagrams. In *Proc. International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *LNCS*, pages 280–294. Springer, 2005.
11. Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, August 1996.
12. Thomas Baar. Non-deterministic constructs in OCL – what does any() mean. In *Proc. 12th SDL Forum*, volume 3530 of *LNCS*, pages 32–46. Springer, 2005.